TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUHO PELTONEN

LIIKENNEOHJAUSJÄRJESTELMÄN SÄÄNTÖKIELI

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunnan
tiedekuntaneuvoston
kokouksessa 9.11.2016

TIIVISTELMÄ

JUHO PELTONEN: Liikenneohjausjärjestelmän sääntökieli
Tampereen teknillinen yliopisto
Diplomityö, 55 sivua
Helmikuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: ohjelmointikieli, täsmäkieli

Tämän diplomityön aiheena on sääntökielen kehittäminen liikennejärjestelmän laitteiden ohjaamista varten. Kieli on tätä sovellusaluetta varten kehitetty täsmäkieli. Tämän lisäksi tässä työssä toteutetaan koodieditori täsmäkielelle.

Ratkaistavana ongelmana tässä työssä on, miten ohjelmointikielen voi suunnitella ja toteuttaa. Lisäksi selvitetään, miten ohjelmointikielelle toteutetaan avustava koodieditori.

Työssä tutkitaan ohjelmointikielien teoriaa yleisesti. Teoriasta käydään läpi sekä suunnittelu- että toteutusnäkökulmat. Suunnittelunäkökulmasta tutkitaan mitä eri ominaisuuksia kielen suunnittelussa tulee ottaa huomioon. Toteutusnäkökulmasta käydään läpi leksikaalinen, syntaksinen ja semanttinen analyysi sekä koodin generointi.

Käytännön osuudessa käydään läpi sääntökielen ja koodieditorin suunnittelua ja toteutusta. Toteutus tehdään Xtext-ohjelmistokehyksellä. Toteutettu kieli ja koodieditori toimii Eclipse-sovelluksessa.

Työn lopputulos saavuttaa tavoitteet. Tämän työn kirjoitusvaiheessa ei toteutettua järjestelmää ole otettu käyttöön, joten tämän tarkempaa tietoa onnistumisesta ei ole olemassa.

ABSTRACT

JUHO PELTONEN: Rule-based language for traffic control system
Tampere University of Technology
Master of Science thesis, 55 pages
February 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Hannu-Matti Järvinen
Keywords: programming language, domain specific language

The subject of this master's thesis is to design and develop a rule-based language for the devices in a traffic control system. The language is a domain specific language developed for this domain. In addition to this, the thesis also includes a code editor for the language.

The problem to be solved is how can a programming language be designed and developed. Also, implementing the code editor is solved.

The programming language theory is researched during thesis in both designing and implementation point of views. The properties of programming language are discussed from the design point of view. From implementation point of view, the lexical, syntatic and semantic analysis is discussed, as is code generation.

In the implementation section of the thesis, language and code editor design and implementation is reported. The implementation is done using Xtext software framework. The resulting work functions in Eclipse-software.

The result of the work is enough to fullfill the target. At the time of this thesis writing, the implemented system is not yet in production, so there are no practical information about the success of the work.

SISÄLLYS

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

DSL	Domain Specific Language, eli täsmäkieli
SQL	Structured Query Language, tietokantojen hallintaan tarkoitettu täsmäkieli
T-LOIK	Eräs Liikenneviraston järjestelmä
Xtext	Sovelluskehys ohjelmointikielien kehittämiseen
Suosituslaskenta	T-LOIK järjestelmän alijärjestelmä
Sääntökieli	Täsmäkieli jota käytetään Suosituslaskenta-järjestelmässä
JVM	Java Virtual Machine, Java virtuaalikone
BNF	Backus-Naur Form, notaatio kontekstittoman kieliopin esittämiseen
LL-jäsennin	Jäsentimen toimintatapa
LR-jäsennin	Jäsentimen toimintatapa
ANTLR	Another Tool For Language Recognition, jäsenningeneraattori
GNU bison	Jäsenningeneraattori
Flex	Työkalu leksikaalisen analyysin toteuttamiseen
Eclipse	Integroitu kehitysympäristö
Xtend	Java-pohjainen ohjelmointikieli

1. JOHDANTO

Ohjelmointikielet ovat tehokkaita työkaluja ilmaisemaan toimintalogiikkaa tietokoneella ajettavaksi. Eri käyttötarkoituksiin kehitetään usein erilaisia ohjelmointikieliä. Kielen käyttötarkoitus voi olla minkä tahansa ohjelmalogiikan esittämistä. Esimerkiksi Java ja Python ovat yleisesti kaikenlaisten ohjelmien toteuttamiseen tarkoitettuja ohjelmointikieliä.

On myös kieliä, jotka on suunniteltu toimimaan vain yhdellä sovellusalueella. Tällaisia kieliä sanotaan täsmäkieliksi (DSL tai Domain Specific Language). Esimerkki täsmäkielestä kielestä on SQL, joka on tarkoitettu tietokannan tietojen hakemiseen ja muokkaamiseen.

Kielen voi toteuttaa kääntäjällä tai tulkilla. Kieli voi olla staattisesti tai dynaamisesti tyypitetty. Eri toimintatavoissa on vahvat ja heikot puolet.

Ohjelmointikielen kirjoittaminen helpottuu, jos sitä tehdään tarkoitusta varten suunnitellulla koodieditorilla. Hyvän koodieditorin on ymmärrettävä kielen rakenteet auttaakseen ohjelmoijaa tehtävässään.

Tämä diplomityö on tehty Liikenneviraston T-LOIK järjestelmän Suosituslaskentatyökalun toteutusprojektiin perustuen. Projektiin kuuluu toteutettava täsmäkieli, koodieditori, hallintatyökalu, sekä palvelinohjelmisto. Projektin toteuttaa Tamperelainen ohjelmistoyritys Bitwise Oy. Diplomityön aihe on rajattu täsmäkielen suunnitteluun ja toteuttamiseen sekä kieltä varten tehtävän koodieditorin toteuttamiseen.

Toteutettavalla kielellä tehdään sääntöjä, jotka määrittävät mittaritietojen perusteella laiteohjauksia. Kieli on käännettävä ja staattisesti tyypitetty. Yleisimmissä käyttötapauksissaan se on virheitä estävä sekä sovellusalueensa sisällä riittävän ilmaisuvoimainen. Näiden ongelmien ratkaisua varten käydään käydään tässä työssä läpi ohjelmointikielen toteutukseen liittyvää teoriatietoa yleisellä tasolla.

Kielen lisäksi tähän työhön kuuluu avustavan koodieditorin toteuttaminen. Itse kielen ja koodieditorin toteutukseen käytetään työssä Xtext-ohjelmistokehystä. Se on

korkean tason työkalu ohjelmointikielien toteuttamiseen.

Luvuissa 3 ja 4 käydään läpi yleisesti ohjelmointikielien toteutusta sekä suunnitteluperiaatteita. Olemassa olevia tekniikoita käydään läpi, sekä tutkitaan niiden soveltuvuutta eri tapauksiin. Luku 5 käsittelee käytännön työtä, jossa toteutetaan kieli sekä koodieditori.

2. TYÖN TAUSTA

Tarve ohjelmointikielen kehittämiselle syntyi Liikenneviraston T-LOIK-järjestelmän Suosituslaskenta-ohjelmassa, johon pitää voida määrittää sääntöjä laitteiden ohjaamiseen.

2.1 T-LOIK

T-LOIK eli tieliikenteen ohjauksen integroitu käyttöliittymä sisältää liikennepäivystäjien tarvitsemat toiminnallisuudet päivystystyöhön. Päivystäjät näkevät yhtenäisen ohjelmiston kaikkiin päivystystyössään tarvitsemiinsa toimintoihin.

T-LOIKiin sisältyy useita eri alijärjestelmiä, jotka koostuvat palvelin- ja käyttöliittymäohjelmistoista. Eri alijärjestelmien palvelimet toimivat keskitetysti yhdessä paikassa saman sovelluspalvelimen sisällä. Yksi näistä alijärjestelmistä on Suosituslaskenta.

2.2 Suosituslaskenta

Suosituslaskenta tuottaa ohjaussuostuksia erilaisille liikenneohjausjärjestelmien laitteille. Ohjaussuositus on laitekohtainen suositus, jonka perusteella laitetta voidaan ohjata. Suosituslaskenta ei ohjaa mitään laitteita suoraan, vaan se lähettää ohjaussuosituksia ohjausjärjestelmälle, joka myös on osa T-LOIK:ia.

Suosituslaskenta sisältää palvelin- ja käyttöliittymäohjelmat. Palvelin toimii samalla alustalla kuin muutkin T-LOIK-palvelimet. Suosituslaskennan käyttöliittymä sisältää hallintatoiminnallisuudet sekä koodieditorin. Hallintaominaisuuksissa hallitaan sääntöjen suoritusympäristöön liittyviä tietoja.

Tämä diplomityö keskittyy koodieditoriin, sekä itse ohjelmointikieleen, joka kehitettiin suosituslaskentaa varten. Suosituslaskennan ohjelmointikieltä kutsutaan sääntökieleksi.

2.3 Sääntökielen suoritusmalli

Sääntökielelle kirjoitettuja sääntöjä suoritetaan Suosituslaskenta-palvelimella. Palvelin on Java-kielellä kirjoitettu ja toimii Java-virtuaalikoneessa (JVM). Niinpä sääntöjä on voitava ajaa helposti JVM:ssä.

Ohjattavat laitteet on jaettu ohjausjaksoihin. Ohjausjakso sisältää yhden tai useamman laitteen. Laite voi kuulua myös useampaan ohjausjaksoon.

Jokaista ohjausjaksoa kohden tuotetaan ohjaussuosituksia usealla eri prioriteetilla. Eri prioriteetteilla on metatietoja, joiden perusteella niitä voidaan kytkeä Ohjausjärjestelmässä automaatille, pois päältä tai ehdottavalle. Automaattinen ohjaus tarkoittaa, että korkeimman prioriteetin ohjaussuositus ohjaa laitetta ilman ihmisen valvontaa. Pois päältä -kytketyn prioriteetin ohjaussuositukset hylätään kokonaan. Ehdottava tarkoittaa tilannetta, jossa ohjaussuositus ei suoraan ohjaa laitetta, vaan kysyy päivystäjältä vahvistuksen suositukselle. Suosituslaskennassa voi olla sääntö jokaiselle ohjausjakso-prioriteetti-parille.

Sääntö voi suorituksen aikana tuottaa ohjaussuosituksen kaikille ohjausjakson laitteille tai vain osalle laitteista. Sääntö voi olla myös ohjaamatta mitään laitetta. Kun korkean prioriteetin sääntö jättää jonkin laitteen ohjaussuosituksen tuottamatta, voi alemman prioriteetin säännön ohjaussuositus aiheuttaa ohjauksen.

2.4 Koodieditorin toiminnalliset vaatimukset

Sääntökielen koodieditorin toiminnalliset vaatimukset olivat projektin alkuvaiheessa seuraavanlaiset:

- syntaksiväritys
- virheiden näyttö
- koodiehdotukset.

Syntaksiväritys tarkoittaa koodieditorissa koodin avainsanojen ja muiden rakenteiden esittämistä eri väreillä. Virheiden näytöllä tarkoitetaan virheiden visualisointia koodieditorissa suoraan koodin seassa. Koodiehdotus tarkoittaa sitä, että editori osaa ehdottaa mahdollisia koodinpätkiä, joita erilaisissa tilanteessa voidaan kirjoittaa.

Lisää toiminnallisuuksia on oltava mahdollista kehittää helposti tulevaisuudessa.

2.5 Kielen vaatimukset

Toteutettavalle ohjelmointikielelle asetettiin ennen suunnittelua minimivaatimukset.

2.5.1 Aritmeettiset ja loogiset operaatiot

Kielen on tuettava yleisimpiä aritmeettisia operaatioita. Näitä ovat yhteen- vähennys- kerto-, sekä jakolaskut (“+”, “-”, “*” ja “/”-merkit). Operaatioiden evaluointijärjestyksen pitää vastata yleistä käytäntöä, jossa kerto- ja jakolaskut suoritetaan ennen yhteen- ja vähennyslaskuja. Numerosta on saatava vastaluku käyttämällä “-”-etumerkkiä.

Aritmeettisten operaatioiden lisäksi on tuettava vertailuoperaatioita. Vertailuoperaatioilla pitää saada tietoon yhtäsuuruus sekä järjestys. Vertailuja voi yhdistellä loogisilla *ja*- sekä *tai*-operaattoreilla. Looginen *ja*-operaattori suoritetaan ennen *tai*-operaattoria. Loogisen totuusarvon voi kääntää *ei*-etuliitteellä.

Aritmeettisia ja loogisia operaatioita pitää voida ryhmitellä sulkeiden eli *“(”* ja *“)”* -merkkien avulla.

2.5.2 Vakiot ja muuttuvan tiedon lukeminen

Kielen on tuettava symbolisten vakioiden asettamista ohjelmakoodissa. Vakioihin pitää voida viitata myöhemmin samassa koodissa. Lisäksi kielellä pitää voida viitata koodin ulkopuolella määriteltyihin vakioarvoihin käyttämällä ennalta tiedossa olevaa nimeä.

Kielellä pitää voida käsitellä muuttuvaa mittaustietoa. Mittaustietoon viitataan yhdistämällä mittalaitteen nimi sekä tietoa vastaavan ominaisuuden nimi.

2.5.3 Luettavat ja ohjattavat laitteet

Kielen on tuettava erilaisia luettavia ja ohjattavia laitteita.

Tässä vaiheessa tuettuja luettavia laitteita ovat tiesääasemat ja liikenteen automaattiset mittauspisteet. Kielen on kuitenkin taivuttava muunkinlaisen tiedon käsittelyyn ilman suuria muutoksia, sillä tulevaisuudessa voi olla muunlaista tietoa saatavilla säännön suoritusta varten.

Sama vaatimus koskee ohjattavia laitteita. Alussa olevia ohjattavia laitteita ovat erilaiset varoitus- ja rajoitustaulut, mutta tulevaisuudessa on oltava mahdollisuus liittää uudenlaisia ohjattavia laitteita järjestelmään.

2.5.4 Ehtolauseet

Sääntökielellä pitää voida kirjoittaa ehtolauseita. Ehtolauseelle voi antaa ehdon, sekä kaksi suoritettavaa lohkoa, kun-lohko ja muuten-lohko. Kun-lohko suoritetaan ehdon ollessa tosi, ja muuten-lohko suoritetaan ehdon ollessa epätosi. Lohko sisältää joukon laiteohjauksia tai lisää ehtolauseita. Kieli ei sisällä ehtolauseiden lisäksi muita ohjausrakenteita, kuten silmukoita.

2.5.5 Funktiot

Sääntökielellä pitää voida määritellä uudelleen käytettäviä funktioita. Funktiot ottavat sisään parametreja ja palauttavat niiden perusteella totuusarvoja. Funktioita voi käyttää ehtolauseissa. Rekursiiviset funktiokutsut ovat kielessä kiellettyjä.

2.5.6 Suorituskyky

Sääntöjä on kyettävä suorittamaan riittävän nopeasti. Säännöissä ei ole silmukkarakenteita, jolloin suorituskyky keskittyy usean nopeasti suoritettavan säännön tehokkaaseen ajamiseen. Ohjattavia laitteita on vähintään satoja. Järjestelmän on keskitetysti tuotettava näille suosituksia eri prioriteeteilla ja eri ohjausjaksoille melko nopeaan tahtiin, vähintään useita kertoja minuutissa.

Suorituskyky on otettava huomioon kielen toteutuksessa. Säännöt on voitava suorittaa riittävän nopeasti.

3. OHJELMOINTIKIELET

Ohjelmointikieli on ohjelmoijan työkalu logiikan määrittämiseksi tietokoneelle.

3.1 Kielen suunnittelu

Kielen suunnitteluvaiheessa on oleellista selvittää minkälaisen semantiikan kieli vaatii ja millainen on sen syntaksi.

3.1.1 Kielen semantiikka

Kielen semantiikka tarkoittaa sen eri toimintojen merkitystä. Se määrittää ominaisuudet kielelle. [2, sivu 48]

Käyttötarkoituksen perusteella kielelle valitaan erilaisia vaatimuksia, jotka kielen on täytettävä ja ominaisuuksia, jotka kielen on toteutettava. Yleisesti ottaen kieltä on voitava helposti käyttää käyttötarkoituksissaan. Jos kielestä puuttuu ominaisuuksia, joita tarvitaan sen yleisissä käyttötapauksissa, on sillä hankala tai mahdoton saavuttaa haluttua lopputulosta. Jos ominaisuuksia on toisaalta liikaa, voi kielen luettavuus heikentyä. [2, sivu 23]

Kielellä on hyvä ilmaisuvoima, jos sillä pystytään ilmaisemaan tehokkaasti tarvittavia toimintoja. Jos ilmaisuvoima ei ole riittävä, joutuu kielen käyttäjä tekemään ylimääräistä työtä saadakseen aikaan haluamansa toiminnon.

Kielen toiminnot ovat luotettavia, jos sillä on helppo välttää virheiden tekeminen. Luotettavuutta auttaa, jos kielen toteutus pystyy analysointivaiheessa löytämään virhetilanteet ja ilmoittamaan niistä.

Tietotyypeillä voidaan vaikuttaa miten kielen eri muuttujat toimivat eri tilanteissa. Erilaisia tietotyyppejä voivat olla esimerkiksi kokonaisluku ja merkkijono.

3.1.2 Kielen tietotyypit

Ohjelmointikielen tyyppijärjestelmä määrittää muuttujille tehtävät mahdolliset operaatiot sekä muuttujien vuorovaikutuksen keskenään. Tyyppijärjestelmä estää operaatiot tyypeiltään epäyhteensopivien muuttujien välillä. [2, sivu 69]

Ohjelmointikielessä voi olla mahdollista muuntaa tietotyyppejä toisiksi implisiittisesti. Tämä helpottaa koodin kirjoittamista. Esimerkiksi seuraavassa C-kielen lauseessa muuntaa kääntäjä kokonaisluvun *1* automaattisesti liukuluvuksi *1.0*, jolloin suoritetaan kahden liukuluvun yhteenlasku:

```
tulos = 1 + 2.0;
```

Toisaalta tällainen tyyppimuunnos voi aiheuttaa tilanteen, jossa ohjelmointikielen käyttäjä käyttää väärää tietotyyppiä tietämättään, koska kääntäjä ei ilmoita tyyppivirhettä.

Tyyppijärjestelmän tarkistelut voi suorittaa käännösaikaisesti eli staattisesti tai ajonaikaisesti eli dynaamisesti.

3.1.3 Dynaaminen tyypitys

Dynaamisessa tyypityksessä muuttujien väliset toiminnot validoidaan ohjelman ajon aikana. Tämä on yleinen toimintatapa tulkatuissa kielissä.

Dynaaminen tyypitys mahdollistaa ohjelmakoodin vapaamman kirjoittamisen, sillä kääntäjän ei tarvitse tietää muuttujien tyyppejä. Riittää, että muuttujat pystyvät suorittamaan ohjelmakoodissa määritellyn toiminnon. Esimerkiksi seuraava funktio Python-kielellä toimii odotetusti sekä kokonaisluvulla, että liukuluvulla:

```
def summa(a, b):
    return a + b
```

Huonona puolena dynaamisessa tyypityksessä on, että ohjelmoijan virheet muuttujien käytön suhteen tulevat ilmi vasta ohjelman ajon aikana. Jos kääntäjällä olisi tiedossa muuttujien tyypit, voisi se suorittaa toimintojen validoinnin jo käännöksen aikana.

3.1.4 Staattinen tyypitys

Staattisessa tyypityksessä tyyppijärjestelmä validoi tiettyjen ominaisuuksien toimimisen käännösvaiheessa ennen ohjelmakoodin suoritusta. Kääntäjä tietää muuttujien tietotyypit, joten se voi tarkistaa mitkä operaatiot ovat mahdollisia millekin muuttujalle. Tämä parantaa kielen luotettavuutta ja estää ohjelmoijaa tekemästä virheitä.

Kun muuttujien tyypit ovat tiedossa ennen suoritusta, on ohjelmointia avustavien työkalujen kehittäminen helpompaa. Koodieditori voi esimerkiksi näyttää sallitut operaatiot, eli editori voi tukea koodin täydennystä.

Staattinen tyypitys mahdollistaa optimaalisemman suorituksen dynaamiseen tyypitykseen verratuuna. Suoritusvaiheessa ei ole tarvetta tarkistaa onko operaatio mahdollista tehdä tai miten operaatio tehdään kyseiselle muuttujalle, sillä suoritettava operaatio on mahdollista tietää koodin analysoinnin jälkeen.

Staattinen tyypitys voi toimia koodin dokumentaationa. Muuttujan tarkoitus on helpompi ymmärtää, jos sen tyyppi on aina tiedossa lukijalle. Funktion käyttäminen on helpompaa, kun on heti tiedossa, minkä tyyppiset parametrit se ottaa ja minkä tyyppisen arvon se palauttaa.

Ongelmallinen tilanne staattisella tyypityksellä tulee ilmi, kun halutaan luoda funktio, joka toimii erityyppisillä parametreilla ja paluuarvoilla. Dynaamisella tyypityksellä tätä ongelmaa ei ole.

3.1.5 Syntaksi

Kielen syntaksi eli kielioppi määrittää miten erilaisia tekstin osia voi liittää mahdollisen ohjelman muodostamiseksi. Syntaksi vaikuttaa merkittävästi koodin luettavuuteen ja kirjoitettavuuteen.

Syntaksin kuuluu tukea kielen käyttötarkoitusta ja semantiikkaa. Syntaksilla olisi hyvä olla helppo ilmaista vähintään yleisimmät kielen käyttötapaukset.

3.1.6 Avainsanat

Ohjelmointikielessä avainsanat ovat ennaltamääriteltyjä merkkijonoja, joilla on erityinen merkitys kielen syntaksissa.

Avainsanalla voi olla jokin tietty merkitys, kuten esimerkiksi Java-kielen avainsanalla *if*, joka aloittaa kielen ehtolauseen. Avainsanalla voi olla kielessä myös monta merkitystä, kuten esimerkiksi Java-kielen *final*, jota voidaan käyttää vakiomuuttujan määrittelyyn tai periytymättömän metodin tai luokan määrittelyyn.

Kuvaavien avainsanojen käyttäminen auttaa kielen ymmärrettävyyttä. Luettavuutta auttaa myös, jos siinä käytetyt merkit ja sanat ovat yleisesti käytössä muissa ohjelmointikielissä. Tästä on hyötyä varsinkin, jos kielen käyttäjällä on ennestään kokemusta ohjelmoinnista. [7]

Jos avainsanoja joudutaan koodissa käyttämään liikaa, tai ne ovat liian pitkiä, hidastuu koodin kirjoittaminen ja lukeminen sen liiallisen monisanaisuuden takia. Esimerkiksi java kielessä funktion määrittely voisi näyttää seuraavalta:

```
public class Esimerkki {
    public static int summa(int a, int b) {
        return a + b;
    }
}
```

Toisaalta esimerkiksi Scala-kielessä vastaava määritys saadaan aikaiseksi seuraavanlaisella koodilla:

```
object Esimerkki {
    def summa(a: Int, b: Int) = a + b
}
```

3.2 Kielisovellusten tyyppejä

Ohjelmointikielen toteuttavilla sovelluksilla on erilaisia käyttötarkoituksia ja toimintamalleja [1, sivu 21]. Kääntäjä ja tulkki ovat näistä yleisimpiä tapauksia.

3.2.1 Kääntäjä ja tulkki

Usein ohjelmointikielen toteuttava ohjelma lukee koodia ja tuottaa suoritettavaa kieltä. Tällaista toteutusta kutsutaan kääntäjäksi. Esimerkiksi ohjelmointikielet C ja Haskell ovat yleensä toteutettu kääntäjän avulla.

Kääntäjän on toimiakseen suoritettava syötteen jäsennys, semanttinen analyysi, sekä koodin generointi. Monet kääntäjät suorittavat lisäksi myös erillisen optimointivaiheen, jossa tuotettavasta koodista pyritään saamaan mahdollisimman tehokasta. Kääntäjä tuottaa tyypillisesti tietokoneen suorittimen ymmärtämää konekieltä. [1, sivu 22]

Toinen keino suorittaa ohjelmaa on käyttää tulkkia. Tulkki lukee koodia, jonka se voi suorittaa ilman konekielen tai muun kielen tuottamista. Tulkki suorittaa syötteen jäsentämisen ja semanttisen analyysin. Suoritus voi tulkissa tapahtua semanttisen analyysin aikana, mutta joissain toteutuksissa tulkki tuottaa ensin konekielisen version tulkattavasta ohjelmasta. Tulkin lopputuloksena on ohjelman suoritus, eli tulkki antaa ulos sen mitä tulkattava ohjelma tulostaa.

On myös yleistä yhdistää kääntäjän ja tulkin toiminnallisuutta siten, että kääntäjävaihe tuottaa välitason kieltä, jonka tulkkiosa suorittaa. Esimerkiksi Java-kielen yleisin toteutus toimii näin. Koodi käännetään ensin Javan tavukoodiksi, jonka sitten Javan virtuaalikone (JVM) suorittaa.

3.2.2 Muuntaminen toiseksi korkean tason kieleksi

Eräs mahdollinen toteutustapa on lukea korkean tason kieltä ja sitten kirjoittaa tuloksena toista samantasoista kieltä. Tällaista ei yleensä kutsuta kääntäjäksi vaikka tuloksena onkin mahdollisesti suoritettavaa kieltä ja kielen toteutus sisältää samat vaiheet kuin kääntäjän toteutus.

3.2.3 Kieltä ymmärtävät työkalut

Kielen ympäristön toteutuksen tavoitteena ei aina ole lainkaan suorittaa syötekieltä tai tuottaa siitä suoritettavaa muotoa. Esimerkiksi monien nykyisten integroitujen kehitysympäristöjen koodieditori analysoi kirjoitetun kielen ja esittää varoituksia ja virheitä.

Samoissa työkaluissa on yleensä myös mahdollisuus etsiä viittauksia ja nimetä symboleja uudelleen. Nämä molemmat vaativat kielen semantiikan ymmärtämistä, jota varten on toteutettava sekä syntaksinen että semanttinen analyysi.

3.3 Kääntäjän yleisrakenne

Ohjelmointikielen toteuttavan sovelluksen voi jakaa osiin seuraavanlaisesti [1, sivu 21]:

- jäsentäjä
- semanttinen analysoija
- generaattori.

3.3.1 Jäsentäjä

Toteutuksen lukuosa vastaanottaa syötteen, josta se rakentaa välitason esitysmuodon (intermediate representation) käymättä tarkemmin läpi syötteen merkitystä. Toisin sanoen tämä vaihe ymmärtää kielen syntaksin ja muuttaa sen helpommin käsiteltävään muotoon.

Useimmiten syöte on tekstipohjaista kieltä, mutta se voi olla muunkin muotoista. Esimerkiksi visuaalisissa ohjelmointiympäristöissä syöte voi olla binäärimuotoinen esitys graafisesta rakenteesta.

Lukija tuottaa välitason esityksen, joka on tyypillisesti jäsennyspuu (parse tree) tai abstrakti syntaksipuu (abstract syntax tree).

3.3.2 Semanttinen analysoija

Semanttinen analysoija vastaanottaa jäsentäjän tuottaman välitason esityksen. Semanttinen analysoija selvittää, mitä kukin syötteen symboli tarkoittaa. Kielestä ja aiheesta riippuen se voisi olla vaikkapa muuttujan tai funktion nimi.

Semanttinen analysoija tuottaa kääntäjässä syötteen pohjalta toisen välitason esityksen, jossa tuntemattomien symbolien tarkoitus on selvitetty niin hyvin kuin kielen toteutukselle on tarpeellista. Tulkissa semanttinen analysoija voi suorittaa ohjelmakoodin.

3.4 Leksikaalianalyysi

Jäsentäjän voi jakaa kahteen osaan, leksikaalianalyysiin ja syntaksianalyysiin. [2, sivu 32]

Leksikaalinen analyysi jakaa syötetekstin alkioiksi (lexeme). Näiden alkioiden välisistä yhteyksistä ei tässä vaiheessa välitetä, vaan leksikaalinen analyysi tuottaa suoraviivaisesti alkioita siinä järjestyksessä, kun ne syötteessä ilmenevät.

Taulukko 3.1 *Esimerkki Javalla kirjoitetun koodirivin alkioista*

Alkio	Kuvaus
int	varattu sana “int”
x	tunniste
=	yhtäsuuruusmerkki
fn	tunniste
(	avaava sulku
a1	tunniste
,	pilkku
50	kokonaisluku
)	sulkeva sulku
;	puolipiste

Alkioihin liittyy tieto minkälaisesta alkiosta on kyse. Tätä sanotaan alkionimeksi (token).

Omia alkioitaan ohjelmointikielissä ovat tyypillisesti avainsanat, numerot, merkkijonot, tunnisteet, operaattorit ja erottimet. Kommenttejakin voi ajatella alkiona, mutta ne jätetään tyypillisesti käsittelemättä syntaksianalyysissä.

Esimerkiksi Java-kielisestä koodista *int x = fn(a1, 50);* voidaan tunnistaa taulukon 3.1 mukaiset alkiot.

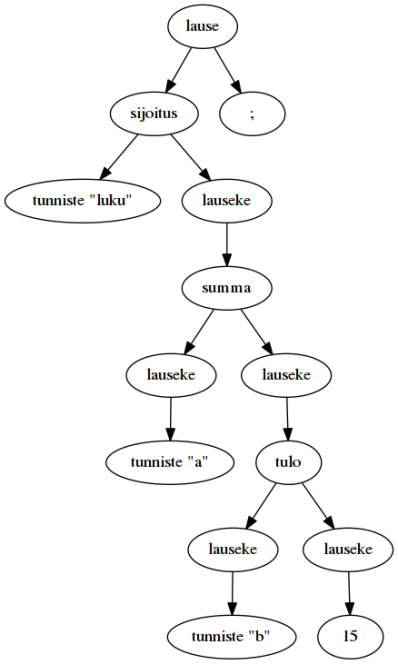
Leksikaalisessa vaiheessa ilmenevät virheet, joissa alkioita ei saada tuotettua. Virheet, joissa alkioiden välinen asiayhteys on väärin eivät ilmene leksikaalisessa analyysissä. Esimerkiksi seuraavat koodinpätkät aiheuttaisivat Java-kääntäjässä virheen leksikaalisesta analyysiä tehdessä:

- *"merkkijono*
- *50x.*

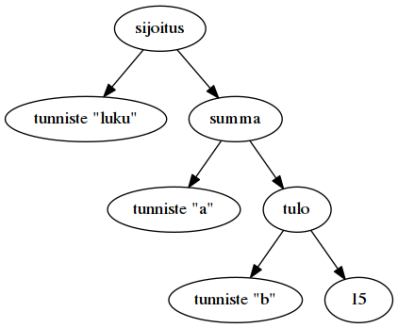
Näistä ensimmäisessä puuttuu merkkijonon lopettava lainausmerkki. Toisessa on on numero, jossa on x-merkki perässä.

3.5 Jäsentimen toteutus

Jäsentäminen (parsing) tarkoittaa kielen syntaksin selvittämistä. Ohjelmointikielen jäsentämisen suorittaa jäsennin (parser), joka luo syötteen perusteella helposti koneellisesti käsiteltävän tietorakenteen. Suorituksen aikana jäsennin tarkistaa myös syötteen mahdollisten syntaksivirheiden osalta. [1, sivu 39]



Kuva 3.1 *Jäsennyspuu*



Kuva 3.2 *Syntaksipuu*

3.5.1 Jäsennyspuu ja abstrakti syntaksipuu

Jäsennyksen lopputuloksena oleva tietorakenne on tyypillisesti abstrakti syntaksipuu. Tämä ei kuvaa kaikkia kieliopin rakenteita, vaan vain semantiikan kannalta oleellisia.

Kieliopin kaikki rakenteet voi jäsentimen sisällä kuvata jäsennyspuulla. Jäsennyspuu vastaa suoraan kieliopin määrittelyn sääntöjä. Usein tämä sisältää kielen käsittelyn kannalta turhaa tietoa.

Esimerkiksi koodin *luku = a + b * 15;* jäsennyspuu on kuvan 3.1 kaltainen. Vastaava abstrakti syntaksipuu on kuvan 3.2 kaltainen.

3.6 Kontekstittomat kielet

Kieli on kontekstiton, jos sen rakenteet eivät riipu muualla olevista rakenteista. Tällaisen kielen kielioppia kutsutaan kontekstittomaksi kieliopiksi tai yhteydettömäksi kieliopiksi (context-free grammar). Kontekstiton kielioppi koostuu joukoista sääntöjä, joilla voidaan esittää kaikki kielen mahdolliset ilmaisut.

Monien nykyaikaisten ohjelmointikielien syntaksi ei ole täysin kontekstista riippumaton. Toisaalta kontekstittoman kielen jäsentämisestä on monissa tapauksissa lyhyt matka kontekstillisten kielien jäsentämiseen. Kontekstillisen kielen jäsentämisessä voidaan joiltain osin käyttää kontekstittomia sääntöjä. Kontekstiriippuvaa tietoa saatetaan tarvita oikean säännön valitsemiseen. [1, sivu 68]

3.7 Kontekstittoman kielen esittäminen

Yleinen tapa kontekstittoman kieliopin sääntöjen esittämiseen on käyttää BNF-notaatiota (Backus-Naur Form). Säännöt voivat koostua toisista säännöistä tai terminaaleista. Terminaali vastaa leksikaalisen analyysin alkiota. Säännöllä voi olla myös monta vaihtoehtoista ilmentymää [2, sivu 34]. Esimerkiksi BNF-notaation ilmaisu

```
<A> ::= "b" <B> | "c"
<B> ::= "x"
```

kertoo miten säännöt A ja B toimivat. Esimerkissä A ja B ovat sääntöjä, ja b, c, x ovat terminaaleja, jotka kuvaavat yksittäistä merkkijonon merkkiä. Sääntö A koostuu vaihtoehtoisesti b-merkistä, jonka jälkeen tulee B-sääntö, tai pelkästä c-merkistä. B-sääntö hyväksyy pelkästään x-merkin. Esimerkin säännöt mahdollistavat yhdessä seuraavat merkkijonot.

- b x
- c.

Muut merkkijonot ovat tämän kieliopin kannalta vääriä, jolloin jäsennin tuottaa virheen.

Säännöt voivat olla rekursiivisia, eli ne voivat sisältää itsensä joko suoraan tai toisen säännön kautta. Tämä mahdollistaa toistuvien rakenteiden ilmaisemisen, sekä sisäkkäiset rakenteet kielessä.

Joskus kieliopin sääntöjä evaluoitaessa päädytään tilanteeseen, jossa on useampi kuin yksi täsmäävä sääntö. Tällöin kielioppi ei ole yksiselitteinen.

3.8 LL-jäsentäminen

Monissa tapauksissa kontekstivapaan kieliopin voi jäsentää rakentamalla jäsennyspuun rekursiivisesti ylhäältä alas vasemmalta oikealle. Tämäntyyppisiä kielioppia sanotaan LL-kieliopiksi (LL grammar) ja jäsennintä kutsutaan LL-jäsentimeksi. [1, sivu 55]

LL-jäsennin käy syötettä vasemmalta oikealle, käyttäen vasemmanpuoleisinta osaa säännön valintaan (Left to right, Leftmost derivation). LL(1)-jäsennin tarkastelee syötettä kerrallaan yhden alkion eteenpäin. LL(k)-jäsennin tarkastelee k-alkiota eteenpäin. Jäsentimen tulevan syötteen tarkastelusta käytetään termiä kurkistus (lookahead). Kurkistuksen määrä riippuu siitä, miten monta alkiota tarvitsee nähdä eteenpäin oikean kielioppisäännön valitsemiseksi.

Yleinen tapa toteuttaa LL-jäsennin on tehdä rekursiivinen laskeutuminen (recursive descent). Rekursiivisessa laskeutumista jokaista sääntöä kohden voidaan tehdä jäsenninrutiini sekä tarkistusfunktio kertomaan täsmääkö sääntö. Kielioppisääntö vastaa melko suoraan ohjelmoitua toteutusta.

Vain osa kontekstittomista kieliopeista on mahdollista jäsentää suoraan LL-jäsentimellä. LL-jäsennin ei voi suorittaa vasenrekursiivista (left recursive) sääntöä. Vasenrekursiivinen sääntö viittaa itseensä ilman, että se ensin viittaa johonkin muuhun. Usein tällaiset säännöt voi kirjoittaa myös ilman vasenrekursiivisuutta. Esimerkiksi seuraava sääntö aiheuttaisi ikuisen silmukan yksinkertaisessa rekursiivisesti laskeutuvassa jäsentimessä, sillä *lauseke*-sääntö yrittää viitata itseensä ilman, että sitä edeltää mikään muu sääntö tai alkio:

```
<lauseke> ::= <lauseke> "+" <numero>
            | <numero>
```

Toisaalta vastaavanlaisen syntaksin voisi kuvata myös seuraavalla säännöllä:

```
<lauseke> ::= <numero> "+" <lauseke>
            | <numero>
```

Tämän lopputuloksena on kuitenkin erilainen jäsennyspuu.

3.9 LR-jäsentimet

Minkä tahansa kontekstivapaan kieliopin voi jäsentää LR-jäsentimellä. Toiminnaltaan LR-jäsennin on monimutkaisempi kuin LL-jäsennin. LR-jäsennin käy syötettä läpi vasemmalta oikealle, käyttäen oikeanpuolimmaista mahdollista joukkoa alkioita säännön valintaan (Left to right, Rightmost derivation). [2, sivu 46]

LR-jäsentäja kokoaa pinon päälle syötealkioita (shift), joita pyritään pelkistämään aina niin, että säännön oikea puoli korvataan säännön vasemmalla puolella (reduce). Pinon rakentamista ja pelkistämistä jatketaan, kunnes syöte on luettu ja pinossa olevaa sääntöä ei voi enää pelkistää.

Merkittävä ongelma LR-jäsentimen toteutuksessa on päättää luetaanko lisää syötettä pinoon vai pelkistetäänkö pinossa olevia arvoja (shift/reduce conflict). Aina pinossa olevista arvoista ja seuraavista syötearvoista ei pystytä yksiseltteisesti tietämään kumpi operaatioista kuuluu suorittaa. Tämä vaikeuttaa LR-jäsentimen toteuttamista.

3.10 Jäsentimen toteuttaminen

Jäsentimen toteutus sisältää yleensä jäsenninsääntöjen ajamisen, sekä abstraktin syntaksipuun luonnin.

3.10.1 LL-jäsentimen toteutus

BNF-muotoiset LL-kieliopit ovat suoraviivaisia toteuttaa tietokoneohjelmalla käyttäen rekursiivista laskeutumista.

Yleisesti ottaen jäsentimessä tarkistetaan mitä olemassa olevaa sääntöä voidaan soveltaa, jonka jälkeen kyseinen sääntö suoritetaan. Koska sääntö koostuu usein toisista säännöistä, käydään lausetta rekursiivisesti syvemmälle, kunnes sääntö koostuu pelkästään yksittäisestä alkiosta.

Otetaan esimerkiksi jäsennin, joka osaa jäsentää listan, jonka sisällä on symboleja tai toisia listoja. Tällaisen syntaksin voisi kuvata seuraavilla BNF-muotoisilla säännöillä:

```
<lista>         ::= "(" <listan-alkiot> ")"
<listan-alkiot> ::= <listan-alkio> <listan-alkiot>
                  | ""
<listan-alkio>  ::= <symboli>
                  | <lista>
<symboli>       ::= "A"
                  | "B"
                  | "C"
```

Nämä säännöt toteuttava yksinkertaistettu pseudokoodi voisi näyttää esimerkiksi seuraavalta:

```
def lista():
    lue('(')
    listan_alkiot()
    lue(')')

def listan_alkiot():
    if seuraavaksi_listan_alkio() then
        listan_alkio()
        listan_alkiot()
    else
        // tyhjä lista

def listan_alkio():
    if seuraavaksi_symboli() then
        symboli()
    else if seuraavaksi_lista() then
        lista()
    else
        virhe()

def symboli():
    lue_symboli()
```

Esimerkkikoodissa *lue* on funktio, joka lukee lukijan tuottaman alkion ja *lue_ symboli* on funktio, joka lukee lukijan tuottaman symboli-alkion, A, B tai C. Funktiot *seuraavaksi_ listan_ alkio*, *seuraavaksi_ lista* ja *seuraavaksi_ symboli* kertovat täsmääkö

seuraavaksi tulevat luetut alkiot kyseiseen sääntöön. Esimerkin tapauksessa tarvitsee kurkistaa yksi alkio eteenpäin, joten kyseessä on LL(1)-jäsentäjä.

Tässä tapauksessa *seuraavaksi_symboli* toteutus tarkistaa, onko seuraava luettu alkio joko *A*, *B* tai *C*. Funktion *seuraavaksi_lista* totetus voisi yksinkertaisimmillaan tarkistaa onko seuraava alkio avaava sulje "(". Funktion *seuraavaksi_listan_alkio* toteutus voi käyttää funktioita *seuraavaksi_symboli* ja *seuraavaksi_lista* apunaan, sillä toisen näistä ehdoista on täsmättävä.

3.10.2 Abstraktin syntaksipuun rakentaminen

Jäsentimen sääntöjä seuraamalla saadaan helposti aikaan syötteen jäsennyspuun, mutta useimmiten on tarvetta abstraktille syntaksipuulle, jota on helpompi käsitellä. Syntaksipuun rakenne rippuu semanttisen analysoijan tarpeista. [1, sivu 92]

Abstraktissa syntaksipuussa poistetaan jäsennyspuuhun verrattuna ylimääräiset sääntösolmut. Esimerkiksi kuvissa 3.1 ja 3.2 näkyy, kuinka abstraktista syntaksipuussa ei ole *lauseke*-solmuja, jotka Java-kielen jäsenninsäännöissä sisältävät vain muita vaihtoehtoisia sääntöjä, mutta eivät itsessään varsinaista tietoa. Myös puolipistesolmu on poistettu, koska sillä ei ole semanttista arvoa. Abstraktia syntaksipuuta rakentaessa käydään siis jäsennyspuun solmut läpi, ja otetaan mukaan vain semanttisesti tärkeät solmut.

3.10.3 Työkaluja jäsentimen tekemiseen

On olemassa useita työkaluja, joilla voi luoda jäsentimen kielioppisäännöistä (parser generator). Nämä työkalut ottavat sisäänsä säännöt tekstimuotoisena, joka usein on jokin BNF-muodon tapainen listaus. Sääntöjen perusteella luodaan jäsennin jossakin ohjelmallisesti suoritettavassa muodossa.

Usein jäsentimen luovat työkalut tuottavat korkean tason ohjelmointikieltä, kuten C- tai Java-kieltä, jonka voi liittää osaksi muuta sovellusta. Esimerkkejä paljon käytetyistä koodia tuottavista työkaluista ovat ANTLR ja GNU bison.

Kielioppisäännöistä jäsentimen tuottaminen on mahdollista myös joillain ohjelmakirjastoilla. Esimerkiksi Haskell-kielen Parsec-kirjasto mahdollistaa jäsentimen toteuttamisen korkean tason jäsenninsäännöillä. [5]

3.10.4 ANTLR

ANTLR (ANother Tool for Language Recognition) on LL-pohjainen jäsentimen toteuttava työkalu. Sitä voi tuottaa ohjelman teksti- ja binääritiedon jäsentämiseen.

Versio 4 työkalusta laajentaa LL(*)-jäsentimen toimintamallia ALL(*)-jäsentimellä (Adaptive LL(*)). ANTLR:n tuottamalla jäsentimellä on mahdollisuus tehdä kieliopin analyysiä dynaamisesti ennalta määriteltyjen sääntöjen lisäksi. [3, sivu 15]

ANTLR-työkalun versio 4 kykenee tuottamaan jäsentimen Java-kielellä. Sen lisäksi ANTLR:ssä on mahdollisuus C#, Javascript ja Python -kielisten ohjelmien tuottamiseen. ANTLR itsessään on kirjoitettu Java-kielellä. ANTRL:n suorittaminen vaatii Java-virtuaalikoneen.

ANTLR tuottama jäsennin kykenee lukemaan suoraan teksti- tai binääripohjaista tietoa ilman esikäsittelyä. ANTLR toteuttaa siis sekä leksikaalisen että syntaksisen analyysin.

LL-jäsennin ei kykene suorittamaan vasenrekursiivisia sääntöjä. ANTLR ratkaisee ongelman muuntamalla säännöt sisäisesti pois vasenrekursiivisesta muodosta. Näin ollen ANTLR:lle on mahdollista määritellä vasenrekursiivisia sääntöjä.

3.10.5 GNU bison

Bison on jäsentimen toteuttava työkalu, jonka idea perustuu LR-jäsentimeen. Se lukee Bison-kielisen kontekstittoman kielioppimäärityksen ja tuottaa sen perusteella C- tai C++-kielisen jäsentimen. Bison on yhteensopivaa UNIX-järjestelmien Yacc-työkalun kanssa. [4]

Bison mahdollistaa C tai C++ -kielen käyttämisen syntaksisäännön valitsemiseen epäselvissä tilanteissa, eli sillä on mahdollista tehdä kontekstista riippuvia sääntöjä.

Bison ei toteuta leksikaalianalyysiä. Alkioiden lukeminen on joko toteutettava itse tai siihen on käytettävä jotain muuta työkalua. Yleinen Bisonin kanssa käytetty työkalu on Flex, joka tuottaa C-kielisen ohjelman leksikaalisista säännöistä.

3.11 Koodin generointi

Koodin generoinnilla voidaan tarkoittaa konekielen tuottamista tai korkeamman tason koodin tuottamista.

Jos halutaan tuottaa konekieltä, suoritetaan tyypillisesti erilaisia optimointivaiheita ennen lopullisen koodin kirjoittamista [1, sivu 366]. Optimoinnin ja konekielen tuottamisen tarkempi tarkastelu on tämän diplomityön ulkopuolella.

Korkeamman tason koodin generoinnissa voidaan optimointivaihe jättää tekemättä, koska tuotetun kielen toteutus suorittaa tyypillisesti itse optimointivaiheen. Korkean tason koodigenerointi onnistuu toteuttamalla abstraktin syntaksipuun jokaiselle solmutyypille generointifunktio, joka mahdollisesti kutsuu muiden solmutyyppien generointifunktioita. Näin abstraktista syntaksipuusta saa generoitua koodin kutsumalla juurisolmun generointifunktiota. [1, sivu 303]

4. TÄSMÄKIELET

Täsmäkieli (DSL eli Domain Specific Language) on tietylle sovellusalueelle suunniteltu ohjelmointikieli. Täsmäkielillä ei yleensä tehdä kokonaisia sovelluksia, vaan niitä käytetään muiden kielien rinnalla.

4.1 Täsmäkielen määritelmä

Täsmäkielissä on vähän ominaisuuksia, jolloin niiden ilmaisuvoima on sen verran rajoittunut, että ne eivät sovellu yleiskäyttöisiksi ohjelmointikieliksi. Täsmäkielien ominaisuudet keskittyvät niiden sovellusalueisiin. [6, sivu 33]

Täsmäkieli voi olla ulkoinen (external DSL), jolloin se on täysin eri kieli kuin kehitettävän sovelluksen pääasiallinen ohjelmointikieli. Esimerkiksi SQL on ulkoinen täsmäkieli, jota käytetään sovelluksissa tietokantojen käsittelyyn. Muita ulkoisia täsmäkieliä ovat esimerkiksi säännölliset lausekkeet (regular expressions), Awk ja XML.

Täsmäkieli voi olla myös sisäinen (internal DSL), jolloin se on toteutettu yleiskäyttöisen ohjelmointikielen rakenteilla. Sisäinen täsmäkieli voi yksinkertaisimmillaan olla kokoelma sopivasti nimettyjä funktioita tai metodeita. Esimerkiksi Ruby on Rails -ohjelmistokehys sisältää joukon Ruby-kielen rakenteilla toteutettuja sisäisiä täsmäkieliä, muun muassa tietokannan määrittelyyn ja käyttämiseen sekä web-rajapintojen konfiguroimiseen.

4.2 Täsmäkielien etuja ja haittoja

Täsmäkielillä voidaan parantaa ohjelmistokehityksen tuottavuutta. Sovellusaluetta varten kehitetyt kielen rakenteet tekevät koodista selkeämpää ja virheiden löytämisestä helpompaa. [6, sivu 37]

Täsmäkielien avulla voidaan saada sovellusalueen asiantuntija toteuttamaan ja ymmärtämään sovelluksen toimintalogiikkaa. Yleiskäyttöisillä ohjelmointikielillä tämä ei onnistu yhtä helposti, sillä niiden opettelu on tyypillisisesti vaikeampaa ja eri

alojen asiantuntijoilla ei välttämättä ole ohjelmointiosaamista ennestään. [6, sivu 38]

Huonona puolena täsmäkielien käyttämisessä on se, että jos sovellus on toteutettu usealla eri ohjelmointikielellä, on sovelluksen kehittäjien myös opeteltava käyttämään useita kieliä. Lisäksi täsmäkielen toteuttamiseen voi kulua paljon aikaa. [6, sivu 40]

4.3 Täsmäkielien suunnitteluperiaatteita

Täsmäkielien suunnitteluun ja totetuttamiseen on yleisesti hyväksyttyjä lähetymistapoja ja työkaluja, jotka auttavat kielen semantiikan ja syntaksin kehittämisessä. [7]

4.3.1 Kielen käyttötarkoitus

Kielen käyttötarkoitus tulee ymmärtää riittävällä tasolla ennen kielen suunnittelua. Käyttötarkoituksen perusteella valitaan kielen ominaisuudet.

Käyttötarkoitus määrittää suorituskykyvaatimuksia, joita kieltä suorittavan toteutuksen on täytettävä. Tämä vaikuttaa kielen semanttisiin toiminnallisuuksiin, sillä joitain käsitteitä ei ole mahdollista suorittaa tehokkaasti. Toisaalta jos suorituskyky ei ole tärkeää, on enemmän vapauksia valita tehokkaita käsitteiden ilmaisutapoja kieleen.

4.3.2 Kielen sisältö

Täsmäkielen kuuluu sisältää vain tarvittavat ominaisuudet. Tarvittavia ominaisuuksia ovat sellaiset, joita vaaditaan käyttötarkoituksen käsitteiden ilmaisemiseen tehokkaasti. Ylimääräiset ominaisuudet monimutkaistavat kieltä turhaan. Monimutkaista kieltä on vaikeampi käyttää ja kehittää. Yksinkertaisuus on täsmäkielen tärkeimpiä ominaisuuksia.

Täsmäkielen ei usein tarvitse sisältää paljon yleiskäyttöisiä ominaisuuksia. Tällaisten ominaisuuksien sijaan täsmäkielen kuuluu sisältää juuri käyttötarkoitusta tukevia ilmaisutapoja. Toisaalta jos yhden yleiskäyttöisen ominaisuuden tilalle joudutaan tekemään useita käyttötapauskohtaisia ominaisuuksia, voi yksittäinen yleiskäyttöinen ominaisuus olla parempi vaihtoehto.

4.3.3 Syntaksi

Täsmäkielen syntaksissa kannattaa käyttää samoja ilmaisutapoja kuin muissakin kieliessä. Jos täsmäkielen kirjoittajalla on kokemusta muista ohjelmointikielistä, helpottuu syntaksin opettelu tällöin merkittävästi. Tämä koskee niin avainsanoja kuin rakenteitakin.

Muutenkin täsmäkielin syntaksin suunnittelussa kannattaa ottaa huomioon samat asiat kuin yleiskäyttöisen kielenkin suunnittelussa. Sovellusaluekohtaisesti tästä voi poiketa tarvittaessa.

4.4 Täsmäkielen toteutus

Kielen toteuttamistapa päätetään kun tiedetään minkälainen kieli halutaan. Toteutustapaan vaikuttaa myös siihen käytettävissä olevan ajan määrä. [7]

4.4.1 Valmiiden komponenttien käyttäminen

Kielen toteutusta ei aina tarvitse aloittaa tyhjästä. On mahdollista, että olemassa tarkoitukseen sopiva täsmäkieli, jolloin ei kannata tehdä uutta samanlaista. Lisäksi joskus olemassa olevan kielen voi muokata haluttuun käyttötarkoitukseen pienemmällä vaivalla, kuin uuden kehittäminen olisi.

Joitain osia kielen toteutuksesta voi käyttää uudelleen eri toteutuksissa. Esimerkiksi erilaiset aritmeettiset operaatiot ovat hyvin yleisiä ominaisuuksia kielissä, ja ne toimivat lähes samalla lailla.

4.4.2 Kirjastot ja ohjelmistokehykset

Täsmäkielien kehittämiseen on olemassa työkaluja ja sovelluskehyksiä. Esimerkkejä näistä ovat MontiCore ja Xtext, jotka soveltuvat tekstipohjaisten täsmäkielien kehittämiseen.

Etuna näissä verrattuna ANTLR:n ja GNU Bisonin kaltaisiin jäsenningeneraattoreihin on, että ohjelmistokehykset tuottavat paljon muutakin kuin jäsentimen. Esimerkiksi Xtext auttaa kaikkien kääntäjän osien toteuttamisessa.

5. LIIKENNEOHJAUSJÄRJESTELMÄN SÄÄNTÖKIELEN TOTEUTTAMINEN

Tässä luvussa käydään läpi sääntökielen ensimmäisen version suunnittelua ja toteutusta, sekä koodieditorin toteutus kielelle. Luvussa ei esitetä varsinaisen toteutuksen koodia, vaan toteutusta kuvataan sanallisesti.

5.1 Sääntökielen esittely

Sääntökielellä pystytään kuvaamaan laitteen ohjaaminen syötetietojen perusteella. Syötetieto voi olla jokin vakioarvoinen parametri tai mittalaitteesta luettu tieto. Seuraava koodi on esimerkki sääntökielellä tehdystä säännöstä:

```
sääntö Esimerkki

laite Rajoitusmerkki merkki
lämpö ilman_lämpö

tee

kun ilman_lämpö < 1 C {
  merkki ohjaa 60 km/h
}
```

5.1.1 Sääntöjen suoritus

Ohjattavat laitteet on asetettu kuulumaan ohjausjaksoihin. Laite voi kuulua useampaan kuin yhteen ohjausjaksoon ja ohjausjaksoon tyypillisesti kuuluu useampi ohjattava laite. Ohjausjaksoilla on myös tiesäätietoa ja liikennetietoa tuottavia laitteita, joita voi käyttää säännöissä.

Ohjausjaksoon voi asettaa eri sääntöjä, joilla jokaisella on eri prioriteetti. Eri sääntöjen ei tarvitse ohjata kaikkia ohjausjakson laitteita. Sääntö voi myös jättää ohjaussuosituksen tekemättä.

Kaikki ohjausjakson säännöt suoritetaan ja tieto tehdyistä ja tekemättömistä ohjaussuosituksista välitetään ohjausjärjestelmään, joka tekee päätöksen laitteiden ohjauksesta. Ohjausjärjestelmän päätös perustuu ohjaussuosituksiin sekä järjestelmän omiin asetuksiin. Yleisimmässä tapauksessa laiteohjaus perustuu ohjaussuositukseen, joka valitaan suosituksen tuottaman säännön prioriteetin perusteella.

5.1.2 Tietotyypit

Sääntökieli sisältää useamman tietotyypin, jotka määrittävät minkälaista tietoa esimerkiksi säännön parametri sisältää tai minkälaista tietoa lauseke tuottaa. Seuraavat tietotyypit on määritelty:

- numero
- nopeus
- lämpö
- totuusarvo
- teksti
- laite

Näistä *numero*, *nopeus* ja *lämpö* ovat numeerisia. Niille voi tehdä matemaattisia operaatioita sekä vertailuoperaatioita. Totuusarvoa voidaan käyttää kun-lauseen ehdoissa. Tekstiarvoja voidaan käyttää laitteiden ohjauksiin. Laitetyyppistä tietoa voidaan käyttää ohjauksen määrittelyyn tai laitteen tietojen lukemiseen.

5.1.3 Sääntö

Säännön määritys aloitetaan *sääntö*-avainsanalla. Tätä seuraa säännön nimi sekä parametrien listaus. Parametrit sisältävät tiedon tyypin sekä parametrin nimen. Parametreihin voi viitata säännön sisällä käyttämällä niiden nimeä. Parametrien esittely päättyy *tee*-avainsanaan, jonka jälkeen alkaa säännön runko.

5.1.4 Muuttujavakioiden määritys

Säännöissä voidaan luoda muuttujavakioita. Muuttujavakion luonti sisältää nimen ja arvon. Muuttujavakion luonnin syntaksi on muotoa *<vakion nimi> = <vakion arvo>*. Arvo voi olla lauseke tai teksti. Tyyppiä ei vakionmäärittelyssä kerrota itse, vaan se päätellään vakion arvon asettavasta lausekkeesta automaattisesti. Seuraava koodi on esimerkki muuttujavakioiden määrittelystä:

```
nopeus = 100km/h / 2
jäätyykö_vesi = ilman_lämpö < 0
```

Esimerkissä määritellään nopeus-tyyppinen muuttujavakio arvolla 50km/h ja totuusarvo-tyyppinen muuttujavakio. Niiden arvot määritellään säännön suorituksen aikana.

5.1.5 Laitteiden tietojen lukeminen

Laitteen tiedon voi sääntökielellä lukea syntaksilla *<laitteen nimi>.<ominaisuus>*. Tätä arvoa voidaan käyttää lausekkeen sisällä.

5.1.6 Lausekkeet

Kielellä voi tehdä matemaattisia ja loogisia operaatioita sisältäviä lausekkeita.

Tuetut matemaattiset toiminnot ovat yhteenlasku, vähennyslasku, kertolasku ja jakolasku. Vastaavat operaattorit ovat “+”, “-”, “*” ja “/”. Nämä toiminnot tuottavat numeerisen arvon. Numeerisia suureita voi kääntää vastaluvuksi unaarioperaattorilla “-”.

Loogisia operaatioita ovat erilaiset suureiden vertailut. Näihin käytetään operaattoreita “<”, “>”, “<=”, “>=”, “=” ja “!=”. Loogiset operaatiot tuottavat totuusarvon. Totuusarvoja operaatioita voi yhdistellä *ja*-sekä *tai*-operaattoreilla ja niiden totuusarvon voi kääntää *ei*-operaattorilla.

5.1.7 Laitteiden ohjausten luominen

Laitteen ohjaukseen käytetään ohjaa-avainsanaa. Laiteohjauksen syntaksi on muotoa *<laitteen nimi> ohjaa <ohjauksen arvo>*. Arvon on täsmättävä laitteen ohjauksen tyyppiin.

5.1.8 Ehtolauseet

Ehtolause sisältää sääntökielessä vähintään kun-ehdon ja kun-lohkon. Näiden lisäksi on mahdollisa kirjoittaa mielivaltainen määrä muuten-kun-ehtoja ja -lohkoja, sekä yksi muuten-lohko. Tämä on semanttisesti samanlainen monien yleisesti käytössä olevien kielien if - else if - else -rakenteiden kanssa.

Sääntökielellä kirjoitettuna ehtolause pelkällä kun-ehdolla ja -lohkolla näyttää seuraan esimerkin mukaiselta:

```
kun ilman_lämpö < 1 C tai huono_sää {
  merkki ohjaa 60 km/h
}
```

Esimerkissä *ilman_lämpö* on jokin lämpö-tyyppinen symbolinen arvo ja *huono_sää* on totuusarvo-tyyppinen arvo. Symboli *merkki* on ohjattava laite, joka hyväksyy ohjaukseksi nopeus-tyyppisen arvon. Kun-lohkoa ympäröivät "{" ja "}" -merkit ovat pakollisia.

5.1.9 Olosuhdeluokat

Kielessä voi määritellä uudelleenkäytettäviä totuusarvoisia lausekkeita. Näistä käytetään kielessä termiä olosuhdeluokka. Olosuhdeluokka määritellään omaan tiedostoonsa syntaksilla *olosuhdeluokka <luokan nimi> <parametrilistaus> tee <lauseke>*. Pääasiallinen käyttötarkoitus olosuhdeluokilla on tarkistaa onko jokin tietty olosuhde voimassa. Parametrilista on samaa muotoa kuin säännön määrittelyssä. Olosuhdeluokat vastaavat toiminnaltaan yleiskäyttöisten ohjelmointikielien funktioita.

Olosuhdeluokkaan voi viitata säännöstä tai toisesta olosuhdeluokasta lausekkeen sisällä käyttämällä syntaksia *<olosuhdeluokan nimi>(<ensimmäinen parametri>; <toinen parametri>; . . .)*.

5.1.10 Kommentit

Kielellä on mahdollista kirjoittaa säännön sisään kommentteja, jotka eivät vaikuta semanttisesti mihinkään. Kaksi eri kommentointitapaa on mahdollisia, rivikommentti ja lohkokommentti. Nämä ovat vastaavat kuin esimerkiksi Java-kielessä. Rivikommentti alkaa merkkijonolla // ja jatkuu rivin loppuun. Lohkokommentti alkaa

merkkijonolla /* ja päättyy merkkijonoon */. Lohkokommentti voi olla useamman kuin yhden rivin mittainen.

5.2 Teknologiavalinnat

Sääntökielen ja editorin toteutukseen valittiin Xtext-ohjelmistokehys. Xtext-kääntäjä ja tekstieditori toimivat molemmat Eclipse-alustan päällä.

5.2.1 Tutkitut vaihtoehdot

Ennen varsinaisen työn aloittamista tutkittiin eri työkaluja joilla projektin voi toteuttaa. Selkeästi teknisesti rajoittavana vaatimuksena oli, että sääntöjä on voitava ajaa Java-pohjaisella palvelimella sekä myös suoraan käyttöliittymästä sääntöjen testausta varten.

Suoraviivainen ratkaisu olisi toteuttaa sääntökielen kääntäjä ja editori itse. Kääntäjässä apuna olisi toiminut ANTLR4-jäsenningeneraattori. Kääntäjä voisi tuottaa Java-koodia tai JVM-tavukoodia. Koodieditori olisi tämän jälkeen tehty Java-pohjaisia käyttöliittymäkirjastoja käyttäen.

Toinen tutkittu vaihtoehto oli käyttää Scala-kieltä sääntöjen kirjoittamiseen. Scalalla pystyisi toteuttamaan ohjelmakirjaston, joka tarjoaisi helposti käytettävän rajapinnan. Lopputulos näyttäisi hyvin paljon täsmäkieleltä sääntöjen kirjoittamiseen. Koodieditoreja on Scala-kielelle olemassa useita, jolloin sitä ei olisi tarvinut tehdä itse ainakaan kokonaan. Huonona puolena tässä ratkaisussa on, että Scala-kääntäjä sallii paljon asioita, joita ei sääntökielessä haluta sallia eivätkä virheilmoitukset avusta niin paljon kuin täysin itsetehdyllä kielellä olisi mahdollista.

Lopulta tutkittiin Xtext-ohjelmistokehyksen käyttämistä työssä. Xtext on ohjelmistokehys ohjelmointikielien kehittämiseen. Xtext pystyy luomaan Eclipse-kehitysympäristön sisällä toimivan koodieditorin Xtextillä luodulle kielelle.

Eri vaihtoehtojen tutkimisen jälkeen päädyttiin käyttämään Xtext-ohjelmistokehystä, koska sen avulla toteutettu koodieditori saadaan aikaan nopeasti projektissa määritellyillä ominaisuuksilla.

5.2.2 Xtext

Xtext on ohjelmointikielien ja täsmäkielien kehitykseen tarkoitettu ohjelmistokehys. Xtextillä kieli määritellään kielioppikielen avulla. Tällä saadaan aikaiseksi pohja kie-

lelle, mukaan lukien jäsennin, linkkeri, tyyppitarkastaja, kääntäjä, sekä koodieditori. [8]

Xtext-ohjelmistokehys sisältää paljon valmista toiminnallisuutta määritellyn ohjelmointikielen toteuttamiseen ja käyttämiseen. Xtext mahdollistaa myös kaiken toiminnallisuuden korvaamisen itsetehdyllä toteutuksella, joka sopii paremmin vaatimuksiin.

5.2.3 Xtend

Xtext-ohjelmistokehyksessä käytetään Xtend-kieltä. Xtend on joustava ja ilmaisuvoimainen Javan tapainen kieli, joka kääntyy luettavaksi Java 5 -lähdekoodiksi [9]. Xtend toimii hyvin Xtext-ohjelmistokehyksen sisällä abstraktin syntaksipuun käsittelyssä, validointisääntöjen toteuttamisessa sekä koodin generoinnissa.

5.2.4 Eclipse

Eclipse on yleisesti käytetty Java-virtuaalikoneessa toimiva integroitu kehitysympäristö, joka tukee useita eri ohjelmointikieliä. Eclipse valittiin projektiin käytettäväksi, koska se toimii hyvin Xtext-ohjelmistokehyksen kanssa. Eclipse on myös riittävän joustava, jotta sillä kyetään toteuttamaan koodieditorin lisäksi koko muu hallintakäyttöliittymä suosituslaskentaan liittyen.

Varsinainen koodieditorin sisältävä sovellus perustuu Eclipse RCP (Rich Client Platform) -tekniikkaan. Se mahdollistaa kaikkien Eclipse-ohjelmiston komponenttien käyttämisen siten, että käyttöliittymä on täysin sovelluksen itsensä määrittelemä. Tällöin on mahdollista tehdä sovellus, joka ei ole täysi integroitu kehitysympäristö, vaan yhteen asiaan erikoistunut sovellus täsmäkielen käyttämiselle.

5.2.5 Scala ja Java

Sovelluksen kehittämisessä käytettiin myös Scala- ja Java-kieliä hallintakäyttöliittymän toteuttamiseen sekä Eclipse RCP-sovelluksen konfiguroimiseen. Nämä osuudet projektista ovat kuitenkin enimmäkseen tämän diplomityön ulkopuolella.

5.3 Sääntökielen kielioppisäännöt

Xtext-ohjelmistokehyksessä jäsentimen voi toteuttaa kokonaan Xtext-kielioppikielellä. Jäsentimen toteutuksen saa kirjoittamalla joukon kielioppisääntöjä. Xtext luo sään-

nöistä jäsentimen, joka lukee ohjelmakoodin abstraktiksi syntaksipuuksi.

5.3.1 Säännöt Xtext-kielellä

Xtext-kielioppikielen säännöt muistuttavat BNF-notaatiota. Seuraavassa esimerkissä esitellään sääntö, jonka perusteella voidaan jäsentää Javan int- ja float-tyyppisten muuttujien esittely:

```
MuuttujanEsittely: IntEsittely | FloatEsittely;

IntEsittely: 'int' name=ID;
FloatEsittely: 'float' name=ID;
```

Esimerkin säännöt tallentavat muuttujan nimen kummassakin tapauksessa "name"-nimiseen ominaisuuteen abstraktin syntaksipuun solmuun.

5.3.2 Terminaalit Xtext-kielellä

Xtext-kielioppikielellä voi määritellä terminaaleja säännöllisillä lausekkeilla (regular expression). Xtextin mukana tulee myös joukko valmiiksi määriteltyjä terminaaleja, jotka toimivat useimmissa kielissä suoraan. Valmiita terminaaleja Xtext:ssä ovat esimerkiksi ID, INT ja STRING. Lisäksi lainausmerkkien sisälle voi määritellä kielioppikieleen avainsanaterminaaleja. Sääntökielessä ID-terminaali määriteltiin sallimaan tunnisteissa ääkkösiä, joita Xtext:n oletustoteutus ei sallinut.

5.3.3 Viittaukset Xtext:ssä

Xtext-kielioppikielessä on joitain ominaisuuksia, joita esimerkiksi BNF-notaatiolla ei voi ilmaista. Yksi näistä on mahdollisuus viitata toiseen sääntöön syntaksilla [<*sääntö*>]. Tällöin Xtext luo säännön, joka tunnistaa nimen. Nimen on täsmättävä johonkin aikaisemmin saman näkyvyysalueen sisällä kirjoitettuun säännön ilmentymään. Tällaisella viittauksella saadaan aikaan toiminnallisuus, joka ei ole kontekstivapaa.

Seuraavassa esimerkissä viitataan kahteen *IntEsittely*-sääntöön *IntSumma*-säännössä, jolloin Xtextin tuottama kääntäjä odottaa vastaavien *IntEsittely*-sääntöjen mukaisten määrittelyjen löytyvän samasta näkyvyysalueesta. Viittaus täsmää viitattavan rakenteen *name*-ominaisuuteen.

```
IntEsittely: 'int' name=ID;

IntSumma: vasen=[IntEsittely] '+' oikea=[IntEsittely];
```

Xtextissä viittausten näkyvyysalueet pystyy määrittelemään itse tai sitten voi käyttää oletustoteutusta. Oletustoteutuksessa viitattavan rakenteen on oltava abstraktissa syntaksipuussa jonkin esi-isän lapsi.

5.4 Sääntökielen kielioppi

Sääntökielen kieliopin juurisäännössä tunnistetaan onko tiedoston sisältönä sääntö vai olosuhdeluokka. Sekä sääntö että olosuhdeluokka jatkuvat parametrilistalla.

5.4.1 Parametrilista

Parametrilistan sääntö tunnistaa yhden tai useamman parametrimäärityksen. Parametrimääritys on joko arvon sisältävä parametrimääritys tai laitteen sisältävä parametrimääritys. Nämä erotetaan sillä, että laitteen sisältävää parametrimääritystä edeltää *laite*-avainsana. Arvon sisältävän parametrin määritys koostuu tietotyypin nimestä ja parametrin nimestä. Laitteen sisältävän parametrin määritys koostuu *laite*-avainsanan lisäksi laitetyypin nimestä sekä parametrin nimestä. Kummankin parametrityypin nimeen viitataan Xtext-kielioppikiellellä säännön tai olosuhdeluokan rungossa.

Parametrilistan jälkeen tunnistetaan sekä säännöllä että olosuhdeluokalla *tee*-avainsana. Tämän jälkeen sääntö sisältää lauselistan ja olosuhdeluokka sisältää lausekkeen.

5.4.2 Lauseet

Lauselistan sääntö sisältää vaihtoehtoisesti joko vakiomäärityksen, ehtolauseen tai laiteohjauksen.

Vakiomäärityksen sääntö sisältää vakion nimen, yhtäsuuruusmerkin ja lausekkeen. Vakiomäärityksen nimeen viitataan Xtext-kielioppikielellä lausekkeiden sisällä.

Ehtolauseen sääntö sisältää *kun*-avainsanan, jota seuraa ehdon määrittävä lauseke, jonka jälkeen on kun-lohkon avaava {-merkki. Tämän jälkeen tulee lauselista, jota

seuraa }-merkki, joka lopettaa kun-lohkon. Kun-lohkon jälkeen on valinnainen määrä muuten-kun-lauseita. Nämä vastaavat ehtolauseen alkua, mutta *kun*-avainsanaa edeltää *muuten*-avainsana. Lopuksi ehtolauseen säännössä on vielä vapaaehtoinen muuten-lohko, joka sisältää *muuten*-avainsanan sekä lauselistan {- ja }-merkkien välissä.

Laiteohjauksen sääntö sisältää viittauksen laite-parametriin, *ohjaa*-avainsanan, sekä ohjauksen arvon määrittävän lausekkeen.

5.4.3 Lausekkeet

Lausekkeet koostuvat monesta säännöstä. Päätason lausekesääntö koostuu joko aritmeettisesta negaatiosta, loogisesta negaatiosta tai tavallisesta lausekkeesta. Jokainen näistä koostuu loogisesta lausekkeesta. Aritmeettinen negaatio tunnistetaan edeltävästä miinusmerkistä ja looginen negaatio tunnistetaan edeltävästä *ei*-avainsanasta.

Looginen lauseke sisältää listan vertailulausekkeita eroteltuna *tai*-avainsanalla tai *ja*-avainsanalla. Tieto erottelevasta avainsanasta tallennetaan abstraktiin syntaksipuuhun. Sääntö on kirjoitettu siten, että kun vertailulausekkeita on vain yksi, ei jäsennin tuota loogista lauseketta abstraktiin syntaksipuuhun, vaan jonkin tarkemman tyypin. Sääntökielen kielioppi ei ota kantaa vertailulausekkeiden suoritusjärjestykseen, mutta sääntökielestä generoidussa Java-koodissa vertailulausekkeen erottavat avainsanat muuttuvat suoraan "||" ja "&&" -operaattoreiksi, joilla on määritelty suoritusjärjestys.

Vertailulausekkeen sääntö sisältää vaihtoehtoisesti joko kaksi aritmeettista lauseketta eroteltuna vertailuoperaattorilla tai yhden aritmeettisen lausekkeen. Yhdellä aritmeettisella lausekkeella ei tuoteta abstraktiin syntaksipuuhun vertailulauseketietoa.

Aritmeettinen lauseke sisältää listan summalausekkeita eroteltuna "*"- tai "/"-operaattoreilla. Lista toimii samalla periaatteella kuin loogisen lausekkeen listakin. Suoritusjärjestys on vasemmalta oikealle.

Summalauseke sisältää listan primäärilausekkeita eroteltuna "+"- tai "-"-operaattoreilla. Primäärilauseke on jonkin yksikön sisäävä numeerinen arvo, merkkijono, parametriviittaus, vakioviittaus, olosuhdeluokkaan viittaus tai uusi lauseke sulkumerkkien sisällä. Suoritusjärjestys on vasemmalta oikealle.

Loogisen lausekkeen, aritmeettisen lausekkeen ja summalausekkeen listat on toteutettu eri säännöillä ja vielä tässä kyseisessä järjestyksessä, jotta jäsennin tuottaa

oikeainlaisen abstraktin syntaksipuun. Näiden operaatioden evaluointijärjestyksellä on merkitystä.

5.5 Tyyppijärjestelmä

Kaikilla lausekkeilla on tyyppi. Tyyppijärjestelmän toteutus selvittää mikä lausekkeen tyyppi on. Myös tyyppivirheet paljastuvat tässä kohtaa. Tyyppijärjestelmä on toteutettu sääntökieleen Xtend-kielellä.

Vakioiden tyyppiä ei määrittele säännön kirjoittaja vaan kielen toteutus määrittää sen itse asettavan lausekkeen perusteella. Laiteohjauksen ja olosuhdeluokan oikeellisuuksien tarkastamiseen käytetään tyyppijärjestelmän kertomaa tyyppiä.

Tyyppijärjestelmän toteutus käy lauseketta läpi rekursiivisesti selvittääkseen sen tyypin. Tyyppitieto tallennetaan abstraktiin syntaksipuuhun solmukohtaisesti, jotta samaa tyyppitietoa ei selvitetä uudestaan, jos sitä tarvitaan myöhemmin. Tyyppijärjestelmän palauttama tyyppi lausekkeelle voi olla joko jokin tietotyyppi tai virhe.

5.5.1 Negaatiot

Lausekkeesta tarkistetaan ensin onko se aritmeettinen negaatio, looginen negaatio vai ei kumpikaan. Aritmeettisen negaation tapauksessa palautetaan virhe, jos lausekkeen tyyppi ei ole numeerinen. Loogisen negaation tapauksessa palautetaan virhe, jos lausekkeen tyyppi ei ole totuusarvo. Muissa tapauksissa palautetaan lausekkeen tyyppi sellaisenaan.

5.5.2 Loogiset operaatiot

Loogisen lausekkeen tapauksessa tyyppijärjestelmän tulos on virheellinen, jos jokin sen lausekkeista ei ole totuustyyppiä. Muussa tapauksessa lausekkeen arvo on totuustyyppi.

5.5.3 Vertailulausekkeet

Vertailulausekkeessa vertailtavien arvojen on oltava molempien numeerisia tyyppejä ja sen lisäksi samaa tietotyyppiä. Tällöin lausekkeen tyyppi on totuusarvo. Muuten kyseessä on virhe.

5.5.4 Aritmeettiset lausekkeet

Aritmeettiset operaatiot +, -, *, / toimivat ainoastaan numeerisilla tyypeillä. Muunlaiset tyypit aiheuttavat virheen. Tyyppijärjestelmä käsittelee summa- ja vähennysoperaatiot erilailla kuin kerto- ja jakolaskuoperaatiot.

Summa- ja vähennysoperaatiot toimivat vain samanlaisten numeeristen tyyppien kanssa. Tällöin lausekkeen tyyppinä on sama numeerinen tyyppi kuin sen osilla on. Muuten kyseessä on tyyppivirhe.

Kerto- ja jakolaskuoperaatiot toimivat vain jos vähintään toinen laskettavista arvoista on tavallinen numero, eli molemmat eivät saa olla joko tyyppiä lämö tai nopeus. Jakolaskussa jakajan on oltava tavallinen numero. Muissa tapauksissa tyyppijärjestelmä palauttaa virheen. Jos molemmat laskettavat arvot ovat tavallisia numeroita, on koko lausekkeen tyyppi tavallinen numero. Muuten lausekkeen tyyppi on joko nopeus tai lämpö, riippuen kumman tyyppinen numero esiintyy operaatiossa.

5.6 Semanttiset tarkistukset

Semanttinen virheiden tarkistus toteutetaan Xtext-ohjelmistokehyksellä määrittelemällä validointimetodeja. Validointimetodi ottaa parametrinaan jonkin abstraktin syntaksipuun solmutyypin. Validointimetodissa voi tarkastella abstraktia syntaksipuuta parametrissa saadun solumn ympäriltä, jonka perusteella voidaan tuottaa solmukohtainen validointivirhe.

Validointimetodien tuottamat virheet näkyvät Xtext:n koodieditorissa punaisella alleviivauksella ja virheviestillä.

5.6.1 Lausekkeiden validointi

Lausekkeen validointi selvittää lausekkeen tyypin käyttäen tyyppijärjestelmää. Validointi tuottaa virheen jos tyyppijärjestelmän tuloksena on virhe.

5.6.2 Olosuhdeluokkien validointi

Kielioppimääritys ei ota kantaa olosuhdeluokan lausekkeen tyyppiin. Koska olosuhdeluokka palauttaa aina totuusarvon, on tätä varten olemassa validointimetodi. Metodi tuottaa virheen jos tyyppijärjestelmän mukaan lausekkeen tyyppi on jotain muuta kuin totuusarvo.

Olosuhdeluokkaan viittaamiseen on validointimetodi, joka tarkistaa, että viittauksessa on käytetty oikea määrä parametreja ja että parametrien tyypit vastaavat olosuhdeluokman määrityksessä olevia tyyppejä. Jos parametrit ovat väärin, tuottaa validointimetodi virheen.

5.6.3 Laitteiden ohjauksien validointi

Laitteiden ohjauksen validointi tarkastaa ohjausarvon tuottavan lausekkeen tyypin. Jos tyyppi ei ole sama, kuin mitä laitteen ohjaukseen voi käyttää, tuottaa validointimetodi virheen.

5.6.4 Ehtolauseiden validointi

Ehtolauseiden validointimetodi tarkistaa tyyppijärjestelmän avulla, että ehdon lauseke on tyypiltään totuusarvo. Jos näin ei ole, validointimetodi tuottaa virheen.

5.6.5 Muut validoinnit

Sääntökielen ensimmäisen version toteutukseen ei tehty täysin kattavia validointeja. Tapaukset, joissa on kirjoitettu virheellistä koodia, mutta joita mikään validointimetodi ei huomannut paljastuvat, kun sääntökielen koodista tuotetun Java-koodin kääntäminen epäonnistuu. Nämä tilanteet eivät ensimmäisessä sääntökielen versiossa anna sääntökielen kirjoittajalle selkeää virheviestiä.

5.7 Koodin generointi

Xtextissä koodin generointi onnistuu helpoiten toteuttamalla Xtend-kielellä Generator-luokan. Luokalle toteutetaan *doGenerate*-metodi, joka saa parametrinaan abstraktin syntaksipuun juurisolmun.

Sääntökielen toteutuksessa toteutus luo tiedoston, jonka nimi on muotoa “<säännön nimi>.java”. Tiedoston sisältönä on Java-kielinen luokka, joka toteuttaa yhden metodin, joka sisältää säännön.

Sääntökielen toteutus toimii rekursiivisesti abstraktia syntksipuuta kulkien. Jokaiselle solmutyypille on toteutettu koodin generointi Java-kieleksi.

5.7.1 Luokan generointi

Juurisolmun perusteella luodaan Java-kielinen luokka joko säännölle tai olosuhdeluokalle. Luokalle tuotetaan metodi parametrilistan perusteella, josta tuotetaan jokaista sääntökielen parametria kohden yksi Java-metodin parametri. Säännön metodilla ei ole paluuarvoa ja olosuhdeluokalle tehtävällä metodilla paluuarvo on boolean-tyyppinen.

5.7.2 Säännön tai olosuhdeluokan metodin sisältö

Olosuhdeluokan totuusarvoinen lauseke kääntyy Java-kielelle yhdeksi *return*-lauseeksi. Lauseke käännetään Java-koodiksi. Sääntö sisältää useita lauseita, joista jokaisesta tuotetaan erikseen Java-kielinen lause.

5.7.3 Lauseen generointi

Jokaiselle sääntökielen lausetyypille on oma koodin generointilogiikka.

Vakion määrittely kääntyy Java-kielen muuttujan määrittelyksi muotoon *final <Java-kielinen tyyppi> <muuttujan nimi> = <lauseke>;*. Java-kielinen tyyppi päätellään sääntökielisen lausekkeen tyypistä, jonka selvittää sääntökielen tyyppijärjestelmä.

Ehtolauseen muoto vastaa täysin Java-kielen *if*-lausetta. Ehtolausetta vastaavasta abstraktin syntaksipuun solmusta luodaan *if*-lause, jonka ehdoksi tuotetaan sääntökielen *kun*-lauseen ehtolauseke. Rungoksi tuotetaan aaltosulkeiden { ja } väliin ehtolauseen lauseiden Java-koodi. Vastaavasti muuten-kun-lauseesta tuotetaan *else if* -lause ja *muuten* lauseesta tuotetaan *else*-lause.

Laitteenohjaus-lauseesta tuotetaan Java-kielinen metodikutsu, joka on muotoa *<nimi>.ohjaa(<lauseke>);*.

5.7.4 Tietotyyppien vastaavuudet Java-kielessä

Sääntökielinen tyyppi muuttuu Java-kieliseksi tyypksi taulukon 5.1 perusteella. Jokaiselle laitetyypille on toteutettu Java-kielinen rajapinta, jota sääntökielessä voi käyttää.

Taulukko 5.1 *Sääntökielen tyypin muuttaminen Java-kielen tyypiksi*

Sääntökielen tyyppi	Java-kielen tyyppi
numero	double
nopeus	double
lämpö	double
totuusarvo	boolean
teksti	String
laite	*Laitetyyppikohtainen rajapinta*

Taulukko 5.2 *Sääntökielen lausekkeiden avainsanojen vastaavuudet Javassa*

Avainsana	Java-kielen operaattori
ei	!
ja	&&
tai	||

5.7.5 Lausekkeiden generointi

Sääntökielen avainsanat *ei, ja, tai* kääntyvät taulukon 5.2 mukaisesti. Numeeriset nopeus- ja lämpö-tyyppiset literaalit tuottavat tavallisen Javan numero-literaalin.

Muissa tapauksissa sääntökielen lausekkeiden rakenne vastaa Javan lausekkeiden rakennetta, jolloin lausekkeen abstraktipuun solmuista saa tuotettua Java-koodia helposti.

5.8 Kielen testaus

Sääntökielen kehitysvaiheessa suoritettiin testausta kirjoittamalla sääntökieltä Eclipsen koodieditoriin ja katsomalla sen tuottamia virheilmoituksia. Tämän lisäksi sääntökielelle on toteutettu automaattisia yksikkötestejä.

Yksikkötesteissä yritetään testata kaikki kielen toteutuksen ominaisuudet. Sallituista syötteistä pitää pystyä tuottamaan Java-koodia ja virheellisistä syötetistä pitää tuottaa virhe.

Sääntökielen ensimmäisen version yksikkötestit eivät ole kattavia. Moni toteutettu ominaisuus jää testaamatta. Tältä osin on kehitettävää tulevaisuudessa.

5.9 Sääntöeditori

Xtext tuottaa kielioppikielen määritysten ja validointimetodien avulla toimivan koodieditorin toteutetulle kielelle. Tätä editoria voi laajentaa tai erikoistaa omiin tarkoituksiinsa Xtext-ohjelmistokehyksen rajapinnoilla.

```
sääntö Testisääntö

lämpö lämpötila
laite Nopeusrajoitus kyltti

tee

// Ohjataan lämpöisellä kelillä suuri nopeus
kun lämpötila > 10C {
    kyltti ohjaa 100km/h
}
```

Kuva 5.1 *Koodieditorin syntaksiväritys*

Ensimmäisessä versiossa ei kiireellisen aikataulun vuoksi toteutettu itse sääntöeditoriin toiminnallisuutta, vaan Xtext-ohjelmistokehyksen valmiit ominaisuudet olivat alkuvaiheessa riittäviä.

5.9.1 Syntaksiväritys

Xtext-editorin valmiit syntaksiväritysominaisuudet värittävät avainsanat, numerot, sekä merkkijonot omilla väreillään. Kuvassa 5.1 on esimerkki sääntökielen syntaksivärityksestä.

5.9.2 Virheiden tarkistukset

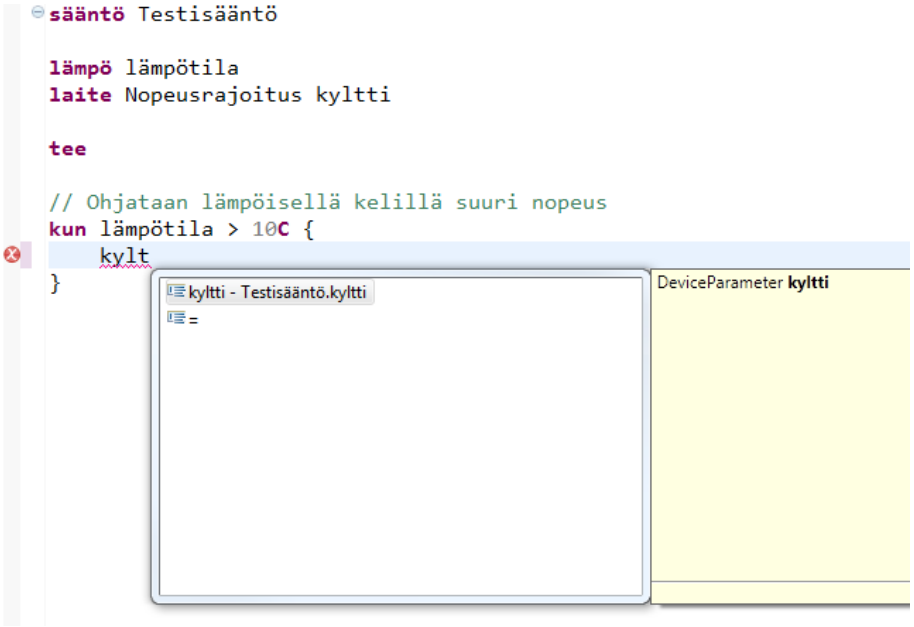
Xtext-ohjelmistokehyksen toteuttama Eclipse-editori näyttää virheet punaisella aleviivauksella. Editorin laidalla näkyy punainen merkintä. Hiiren kursorin päälle siirtämisen jälkeen näytetään virheen kuvausteksti. Editorin virheentarkistus perustuu jäsentimen löytämiin virheisiin sekä itse tehtyihin validointisääntöihin.

5.9.3 Kooditäydennys

Oletustoteutus tukee viittauksien kohdalla koodin täydennystä. Kun kielioppikielellä on määritelty viittaus toiseen kielen rakenteeseen, osaa editori täydentää näiden rakenteiden ilmentymien nimet.

Koodintäydennykseen on mahdollista Xtext-ohjelmistokehyksessä toteuttaa itse kuvaavammat kuvaukset täydennettäville nimille. Tätä ominaisuutta ei toteutettu ensimmäiseen versioon.

Kuvassa 5.2 on esimerkki sääntökielen kooditäydennyksestä. Samassa kuvassa näkyy myös virheentarkistuksen ilmoittama syntaksivirhe.



Kuva 5.2 *Koodieditorin täydennysominaisuus*

5.9.4 Käyttäminen Eclipse-sovelluksessa

Eclipse RCP-sovelluksessa koodieditorin saa auki avaamalla Eclipse-alustan toiminnallisuuksilla projektin ja avaamalla siihen projektiin sääntökieltä sisältävän koodi-tiedoston. Tällöin editori avautuu sovellukseen sen konfiguraation mukaiseen sijaintiin. Tämä määritellään RCP-sovelluksen koodissa.

Toteutusvaiheessa Xtext tuki ainoastaan Eclipse 3.X -rajapintaa. Uusin Eclipse RCP-alusta, nimeltään e4, perustuu uudempaan rajapintaan, jolla toteutettu RCP-sovellus ei toimi Xtextin kanssa. Vanha rajapinta on kuitenkin vielä hyvin tuettu myös uusimmassa Eclipse-alustassa yhteensopivuuskerroksen ansiosta. Tämän takia Eclipse RCP-sovelluksen toteuttaminen käyttäen Eclipse 3 -tekniikkaa ei aiheuttanut ongelmia, koska sääntötyökalun muu osa ei juuri tarvitse Eclipse-alustan rajapintoja muihin tarkoituksiin kuin koodieditorin hallintaan.

5.9.5 Virheenkorjausehdotukset

Xtext-ohjelmistokehys tekee helpoksi korjausehdotusten näyttämisen Eclipsen koodieditorissa. Tuotetulle validointivirheelle voi toteuttaa metodin, joka tekee käyttäjälle ehdotuksen virheen korjaamiseksi. Esimerkiksi laiteohjaus "*nopeusrajoitus ohjaa 60*" on virheellinen, koska lauseke "60" on tyypiltään numero, eikä nopeus. Tässä tapauksessa korjausehdotus voisi muuttaa ohjauksen muotoon "*nopeusrajoitus ohjaa 60km/h*".

Koodieditorin ensimmäisessä versiossa ei toteutettu virheenkorjaustoiminnallisuuksia, joten tämä toiminnallisuus jää kehitettäväksi tulevaisuudessa.

6. TYÖN ONNISTUMISEN ARVIOINTI JA JATKOKEHITYS

Työn aikana suunniteltu ja toteutettu sääntökieli täyttää alkuperäiset toiminnalliset vaatimukset. Kielen ilmaisuvoima on riittävällä tasolla. Kielen perusrakenne sallii erilaisten laitteiden ohjaamisen ja lukemisen, sekä numeeristen parametrien käytön säännön sisällä. Säännön ohjausrakenteet ovat riittäviä käyttötarkoitukseensa. Kielen laajentaminen ensimmäisestä versiosta onnistuu pienellä vaivalla.

Eclipsessä toimiva koodieditori täyttää työn alussa määritelltyt toiminnalliset vaatimukset. Koodieditoriin on helppo toteuttaa lisää toiminnallisuuksia tarpeen mukaan.

Työn toteutukseen valittu Xtext-ohjelmistokehys osottautui hyväksi teknologiavalinnaksi. Sillä kielen kehitys onnistui nopealla aikataululla ja sillä saatu lopputulos oli selkeä rakenteeltaan.

Hieman heikkona puolena Xtext:ssä oli dokumentaation vähyys. Työn tekemisen aikana tietoa piti etsiä Xtext:n lähdekoodista, joka onneksi onnistui helposti, sillä Xtext:n lähdekoodi on avoin. Xtext:n kielioppikielessä oli heikkona puolena sen antamat virheviestit ongelmatilanteissa, jotka eivät virheellisen kieliopin tapauksessa aina sisältäneet lainkaan tietoa ongelman syystä. Nämä ongelmat eivät kuitenkaan estäneet tekemistä missään vaiheessa, vaan vain hidastivat sitä.

Kielen ensimmäistä versiota ei työn kirjoittamisen aikana oteta oikeaan käyttöön. Suosituslaskenta-projektissa ei ole vielä toteutettu sääntöjä suorittavaa osuutta eikä sitä ole otettu tuotantokäyttöön. Ennen käyttöönottoa kieleen todennäköisesti lisätään uusia ominaisuuksia.

Jatkokehitystä ajatellen kieleen on toteutettava operaatioita merkkijonoille, jotta saadaan näytettyä tekstipohjaisen tiedon seassa muuttujien arvoja. Tällaista toimintoa tarvitaan esimerkiksi tilanteessa, jossa tekstiä näyttävän kyltin tekstiin halutaan mukaan lämpömittaustietoa.

Kielen toteutukseen tarvitaan myös lisää validointitoiminnallisuutta. Nykyisien vali-

dointiominaisuuksien virheviestejä on myös muokattava kuvaavimmiksi, jotta niistä selviää helpommin ohjelmointivirheen oikea syy.

Jatkossa kieleen on myös tehtävä lisää automaattitestejä, jotta voidaan varmistua, että kielen toteutukseen ei jatkokehityksen aikana tule virheitä. Ensimmäisessä versiossa automaattitestejä ei ole kattavasti.

Koodieditoriin kehitetään jatkossa koodintäydennykseen konteksista riippuvat kuvaustekstit. Virhetilanteisiin kehitetään korjausehdotustoiminnallisuus, joka ehdottaa korjauksia kirjoitetuun koodiin.

7. YHTEENVETO

Tässä diplomityössä suunniteltiin ja toteutettiin täsmäkieli sekä täsmäkielen koodieditori käyttäen Xtext-ohjelmistokehystä. Ohjelmointikielten teoriaa käytiin läpi suunnittelun ja toteutuksen näkökulmista. Käytännön toteutus kuvattiin sanallisesti.

Toteutettu täsmäkieli oli osa Liikenneviraston Suosituslaskenta-järjestelmän toteutusta. Järjestelmän muut osat käytiin lyhyesti läpi, mutta pääasiassa työ keskittyi kieleen ja koodieditoriin.

Ohjelmointikielien teoriasta tutustuttiin kielten erilaisiin ominaisuuksiin sekä käytiin läpi totetuksen eri vaiheita. Käsiteltyjä toteutuksen vaiheita olivat leksikaalinen analyysi, syntaksinen analyysi, semanttinen analyysi ja koodin generointi. Yleistä teoriaa ja erilaisia jäsentimen toteutustapoja, sekä täsmäkielien suunnitteluperiaatteita käytiin läpi luvuissa 3 ja 4.

Toteutusosuudessa käsiteltiin tieliikenteen laitteiden ohjausta varten kehitetyn täsmäkielen suunnittelu ja toteutus sekä teknologiavalinnat. Toteutetun täsmäkielen sovellusalueena on liikenteenohjauslaitteiden ohjaus. Toteutetusta kielestä käytetään nimitystä sääntökieli. Sääntökielen ominaisuudet on suunniteltu sen käyttäjäryhmää silmälläpitäen, joka ei ole kovinkaan ohjelmointitaitoinen, vaan sen sijaan tuntee oman sovellusalueensa hyvin.

Toteutus tehtiin Xtext-ohjelmistokehystä käyttäen. Sääntökielen ja sitä varten tehdyn koodieditorin suunnittelusta ja toteutuksesta kerrottiin luvussa 5.

Toteutettu työ arvioitiin lyhyesti luvussa 6. Arvioinnin lopputulos oli, että kieli sisältää tarvittavat rakenteet sääntöjen kirjoittamiselle laitteiden ohjausta varten. Käytännön palaute on loppukäyttäjiltä vielä saamatta, mutta toisaalta projekti jatkuu yhä. Jatkokehitystarpeet listattiin myös tässä luvussa.

Diplomityön kirjoitus ja toteutus tehtiin pääasiassa vuoden 2016 syksyn ja loppuvuoden aikana. Toteutettu toiminnallisuus otetaan käyttöön aikaisintaan vuoden 2017 alussa.

LÄHTEET

[1] T. Parr, Language Implementation Patterns - Create Your Own Domain-Specific And General Programming Languages, The Pragmatic Bookshelf, 2010, 389 s.

[2] M. Harsu, Ohjelmointikielet - Periaatteet, käsitteet, valintaperusteet, 2012, 313 s. Viitattu 7.2.2017 Saatavilla: http://www.cs.tut.fi/~popl/nykyinen/Ohjelmointikielet-harsu.pdf

[3] T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, 2013, 322 s.

[4] Bison 2016. WWW-sivu, Viitattu 13.12.2016 https://www.gnu.org/software/bison/manual/bison.html

[5] Parsec - HaskellWiki 2016, WWW-sivu, Viitattu 13.12.2016 https://wiki.haskell.org/Parsec

[6] M. Fowler, Domain-Specific Languages, 2010, 413 s.

[7] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel, Design Guidelines for Domain Specific Languages, 2009, 7 s.

[8] Xtext 2016. WWW-sivu, Viitattu 13.12.2016 http://www.eclipse.org/Xtext/

[9] Xtend 2016. WWW-sivu, Viitattu 13.12.2016 http://www.eclipse.org/xtend/